Chapter 3

Methodology

In this chapter, we present our methodology to answer the research questions enunciated in Subsection 1.1.2. We investigate three research questions. In the first question, we artificially generate WebAssembly program variants and quantitatively compare the static differences between variants. Our second research question focuses on comparing their behavior during their execution. The final research question evaluates the feasibility of using the program variants in security-sensitive environments such as Edge-Cloud computing proposing a multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use an empirical method [**?**], proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

Corpora

Our experiments assess the impact of artificially created diversity in terms of program variants size, static and dynamic differences. The first step is to build a suitable corpus of programs' seeds to generate the variants. Finally, we answer all our research questions with three corpora of diverse and representative programs for our experiments. We build our three corpora in an escalating strategy. The first corpus is diverse and contains simple programs in terms of code size, making them easy to analyze manually. The latter two corpora contain more extensive real-world programs, including one project meant for security-sensitive applications. Finally,

all corpora are considered to come along the LLVM pipeline. We base this decision on the previous experimental work of Hilbig et al. [**?**]. This work shows that approximately 65% of all WebAssembly programs come out of C/C++ source code, and more than 75% if Rust is included. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** : We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library ported to WebAssembly in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Besides, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In Table 3.1 we listed the corpus name, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus.

3.1 RQ1. To what extent can we artifically generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for WebAssembly. We use CROW to generate variants from an original

[1] `http://www.rosettacode.org/wiki/Rosetta_Code`

Corpus	No. modules	No. functions	LOC range	Location
Rosetta	-	303	7 - 150	`http://rosettacode.org/wiki/Rosetta_Code`
Libsodium	102	869	8 - 2703	`https://github.com/jedisct1/libsodium`
QrCode	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust`
Total		3021		

Table 3.1: Corpora description. The table is composed by the name of the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

program, written in C/C++ in the case of Rosetta corpus and LLVM bitcode modules in the case of Libsodium and QrCode. In Figure 3.1 we illustrate the workflow to generate WebAssembly program variants. We pass each function of the corpora to CROW as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated WebAssembly variants.

Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two metrics to quantitatively evaluate RQ1 at the corpus level.

Definition 1 *Program's population $M(P)$: Given a program P and its generated variants, the program's population is defined as.*

$$M(P) = \{v \text{ where } v \text{ is a variant of } P\}$$

Notice that, the program's population includes the original program P.

Definition 2 *Program's unique population $U(P)$: Given a program P and its program's population $M(P)$, the program's unique population is defined as.*

$$U(P) = \{v \ \in \ M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $md5sum(v_i) \neq md5sum(v_j)$. $Md5sum(v)$ is the md5 hash calculated over the byte stream of the program file v. Notice that, the original program P is included in $U(P)$.

Metric 1 *Program's population size $S(P)$: Given a program P and its program's population $M(P)$ according to Definition 1, the program's population size is defined*

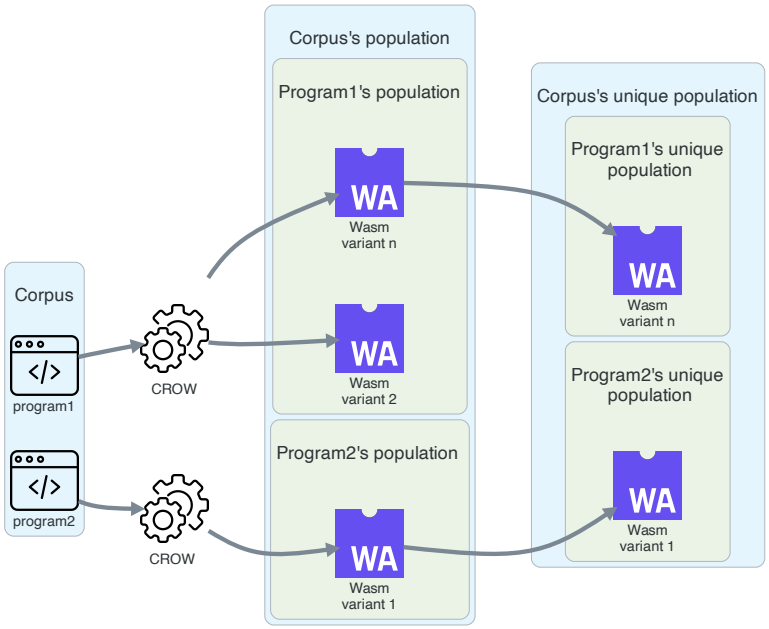

Figure 3.1: The program variants generation for RQ1.

as.

$$S(P) = |M(P)|$$

Metric 2 *Program's unique population size $US(P)$: Given a program P and its program's unique population $U(P)$ according to Definition 2, the program's unique population size is defined as.*

$$US(P) = |U(P)|$$

Metric 3 *Corpus population size $CS(C)$: Given a program's corpus C, the corpus population size is defined as the sum of all program's population sizes over the corpus C.*

$$CS(C) = \Sigma S(P) \; \forall \; P \; \in \; C$$

Metric 4 *Corpus unique population size $UCS(C)$:*

Given a program's corpus C, the corpus unique population size is defined as the sum of all program's unique population sizes over the corpus C

$$UCS(C) = \Sigma US(P) \ \forall \ P \ \in \ C$$

Protocol

To generate program variants, we synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [**?**] against the original program. An enumerative synthesis is a brute-force approach to generate program variants. With a maximum number of instructions, it constructs and checks all possible programs up to that limit. For a simplified instance, with a maximum code size of 2 instructions in a programming language with L possible constructions, an enumerative synthesizer builds all $L \times L$ combinations finding program variants. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all the 60 supported instructions in our synthesizer.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and Libsodium is motivated by the properties listed in Table 3.1. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the $303 + 869 + 1849$ functions in the corpora to CROW, as Figure 3.1 illustrates, to synthesize program variants. We calculate the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus and conclude by answering RQ1.

3.2 RQ2. To what extent are the generated variants dynamically different?

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate our experiments into two fields as Figure 3.2 illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program, and we address it in answering RQ1.

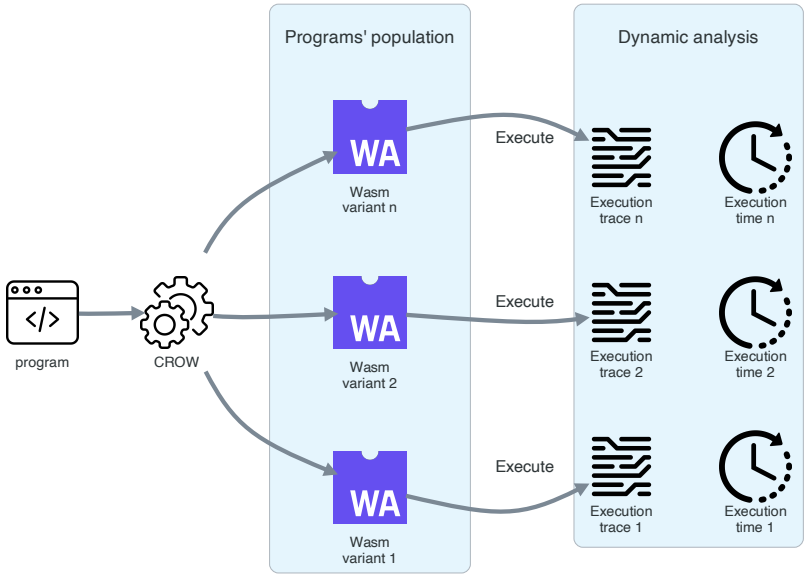

Figure 3.2: Dynamic analysis for RQ2.

With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the original functions from Rosetta corpus described in Section 3 and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the WebAssembly binary to it.

To dynamically compare programs and their variants, we execute each program on each programs' population to collect their execution traces and execution times. We perform fine-grained comparisons by comparing the traces and execution times for all pairs of programs. Therefore, the defined metrics are formulated to support a pairwise comparison strategy. In the following, we define the metrics used to answer RQ2.

Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [**?**] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. In our experiments, we define the traces as the sequence of the stack operations during runtime, *i.e.,* the consecutive list of `push` and `pop` operations performed by the WebAssembly engine during the execution of the program. In the following, we define the $TraceDiff$ metric.

Metric 5 *TraceDiff: Given two programs P and P' from the same program's population, TraceDiff(P,P'), computes the DTW distance between the stack operation traces collected during their execution.*
A TraceDiff of 0 *means that both traces are identical. The higher the value, the more different the traces.*

Moreover, we use the execution time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution time distributions. We define the execution time as follows:

Metric 6 *Execution time: Given a WebAssembly program P, the execution time is the time spent to execute the binary.*

Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (Metric 5). We use SWAM[2] to execute each program and variant to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations. We want to remark that we only collect the stack operation traces due to the memory-agnosticism of our approach to generate variants. Our approach does not change the memory-like operations of the original code.

Furthermore, we collect the execution time, Metric 6, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Withney U test [**?**] in a pairwise strategy.

[2] `https://github.com/satabin/swam`

3.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

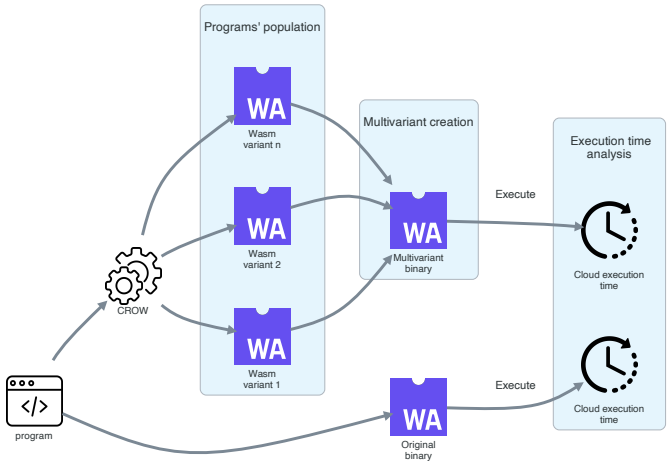

Figure 3.3: Multivariant binary creation and workflow for RQ3 answering.

In the last research question, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries, and we deploy and execute them at the Edge. The process of *mixing* multiple variants into one multivariant binary is an essential contribution of the thesis that is presented in details in [**?**]. RQ3 focuses on analyzing the impact of this contribution on execution times. To answer RQ3, we use the variants generated for the programs of Libsodium and QrCode corpora, we take $2 + 5$ programs interconnecting the LLVM bitcode modules (mentioned in Table 3.1). We illustrate the protocol to answer RQ3 in Figure 3.3 starting from the creation of the programs' population.

Metrics

We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in Metric 6 for the execution time of multivariant binaries.

Protocol

We run the experiments to answer RQ3 on the Edge, executing the multivariant binaries as end-to-end HTTP services. The execution times are measured at the

backend space, *i.e.,* we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header. We do this process for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world.

We collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [**?**] to compare both execution time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Nevertheless, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.